Karlsruhe Institute of Technology

HordeSat: A Massively Parallel Portfolio SAT Solver

SAT 2015, Austin, Texas, USA

Tomáš Balyo, Peter Sanders, and Carsten Sinz | September 22, 2015

INSTITUTE OF THEORETICAL INFORMATICS, ALGORITHMICS II

Definitions

CNF Formula

- A Boolean variable has two values: True and False
- A literal is Boolean variables or its negation
- A clause is a disjunction (or) of literals
- A CNF formula is a conjunction (and) of clauses

$$F = (x_1 \vee x_2 \vee \overline{x_4}) \wedge (\overline{x_3} \vee x_1) \wedge (x_1) \wedge (\overline{x_2} \vee \overline{x_4})$$

Satisfiability

- A CNF formula is satisfiable if it has a satisfying assignment.
- The problem of satisfiability (SAT) is to determine whether a given CNF formula is satisfiable

Introduction

Goal

Design a massively parallel SAT solver that runs well on clusters with **thousands** of processors (for industrial benchmarks)

Results

- HordeSat – new parallel solver
- Experiments with industrial benchmarks with up to 2048 processors
- Significant speedups, especially for hard instances

Parallel Sat Solving

- Explicit Search Space Partitioning
  - classical approach, search space does not overlap
  - each solver starts with a different fixed partial assignment
  - learned clauses are exchanged
  - used in solvers for grids and clusters
- Pure Portfolio
  - modern approach, simple but strong
  - different solver(configuration)s work on the same problem
  - learned clauses are exchanged
  - often used in solvers for multi-core PCs

Parallel Sat Solving

- Explicit Search Space Partitioning
  - classical approach, search space does not overlap
  - each solver starts with a different fixed partial assignment
  - learned clauses are exchanged
  - used in solvers for grids and clusters
- **Pure Portfolio**
  - modern approach, simple but strong
  - different solver(configuration)s work on the same problem
  - learned clauses are exchanged
  - often used in solvers for multi-core PCs

Design Principles

- Modular Design
  - blackbox approach to SAT solvers
  - any solver implementing a simple interface can be used
- Decentralization
  - all nodes are equivalent, no central/master nodes
- Overlapping Search and Communication
  - search procedure (SAT solver) never waits for clause exchange
  - at the expense of losing some shared clauses
- Hierarchical Parallelization
  - running on clusters of multi-cpu nodes
  - shared memory inter-node clause sharing
  - message passing between nodes

Modular Design

Portfolio Solver Interface

```
void addClause(vector<int> clause);
SatResult solve(); // {SAT, UNSAT, UNKNOWN}
void setSolverInterrupt();
void unsetSolverInterrupt();
void setPhase(int var, bool phase);
void diversify(int rank, int size);
void addLearnedClause(vector<int> clause);
void setLearnedClauseCallback(LCCallback* clb);
void increaseClauseProduction();
```

- Lingeling implementation with just glue code
- MiniSat implementation, small modification for learned clause stuff

Diversification

Karlsruhe Institute of Technology

Setting Phases – "void setPhase(int var, bool phase)"

- Random – each variable random phase on each node
- Sparse – each variable random phase on exactly one node
- Sparse Random – each variable random phase with prob. $\frac{1}{\#solvers}$

Native Diversification – "void diversify(int rank, int size)"

- Each solver implements in its own way
- Example: random seed, restart/decision heuristic
- For lingeling we used plingeling diversification

- Best is to use Sparse Random together with Native Diversification.

Clause Sharing

Regular (every 1 second) collective all-to-all clause exchange

Exporting Clauses

- Duplicate clauses filtered using Bloom filters
- Clause stored in a fixed buffer, when full clauses are discarded, when underfilled solvers are asked to produce more clauses
- Shorter clauses are preferred
- Concurrent Access – clauses are discarded

Importing Clauses

- Filtering duplicate clauses (Bloom filter)
  - Bloom filters are regularly cleared – the same clauses can be imported after some time
  - Useful since solvers seem to ”forget” important clauses

Overall Algorithm

Karlsruhe Institute of Technology

The Same Code for Each Process

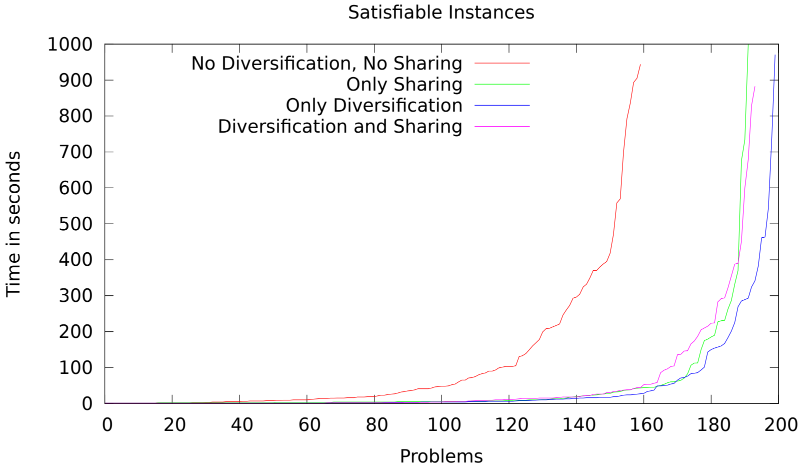
```
SolveFormula(F, rank, size) {
  for i = 1 to #threads do {
    s[i] = new PortfolioSolver(Lingeling);
    s[i].addClauses(F);
    diversify(s[i], rank, size);
    new Thread(s[i].solve());
  }
  forever do {
    sleep(1) // 1 second
    if (anySolverFinished) break;
    exchangeLearnedClauses(s, rank, size);
  }
}
```

Experiments

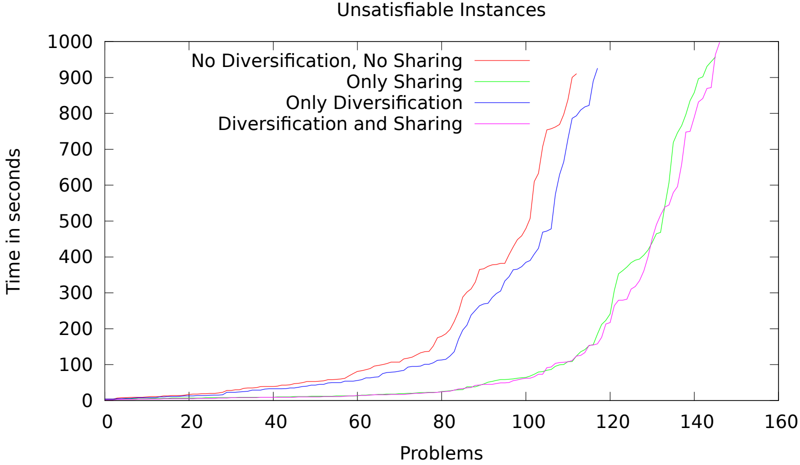
Karlsruhe Institute of Technology

- Benchmarks
  - Sat Competition 2014+2011 Application track instances (545 inst.)
  - Phase Transition Random 3-SAT (200 SAT + 200 UNSAT inst.)
- Computers
  - 128 Nodes of the IC2 cluster
    - each with two octa-core Intel Xeon E5-2670 2.6GHz CPU, 64GB RAM
    - connected by InfiniBand 4X QDR Interconnect
  - In total 256 CPUs and 2048 cores
- Setup
  - Each node runs 4 processes each with 4 threads with Lingeling
  - 1000 seconds time limit (16.7 minutes) for parallel solvers
  - 50000 seconds (13.9 hours) for sequential solvers

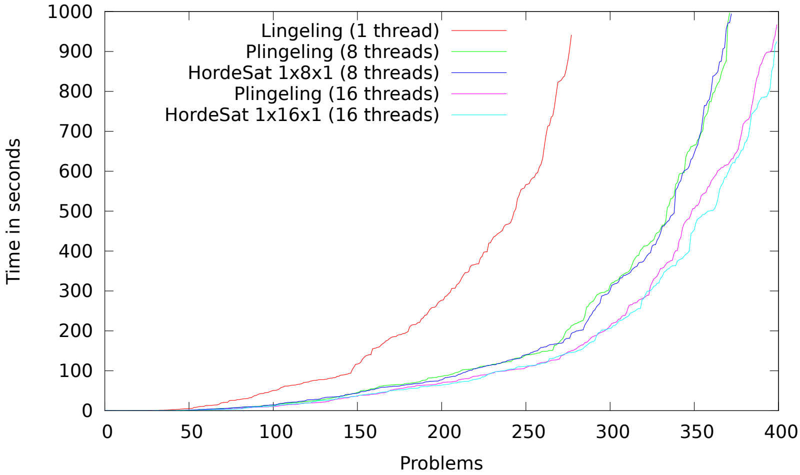
Experiments – Random 3-SAT



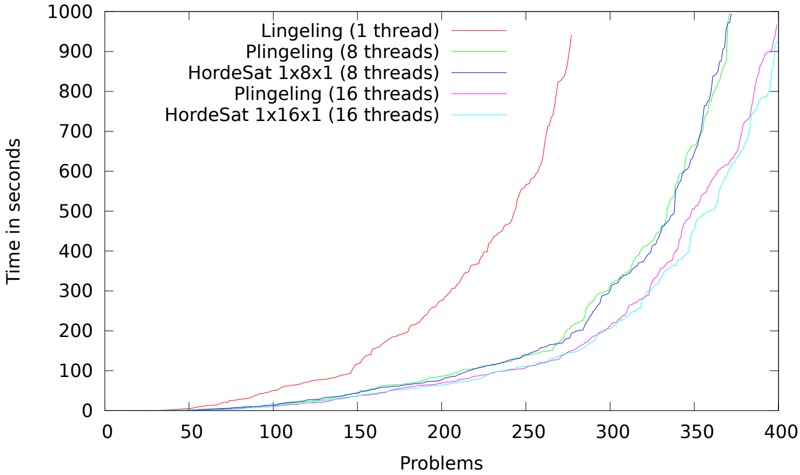
Experiments – Random 3-SAT



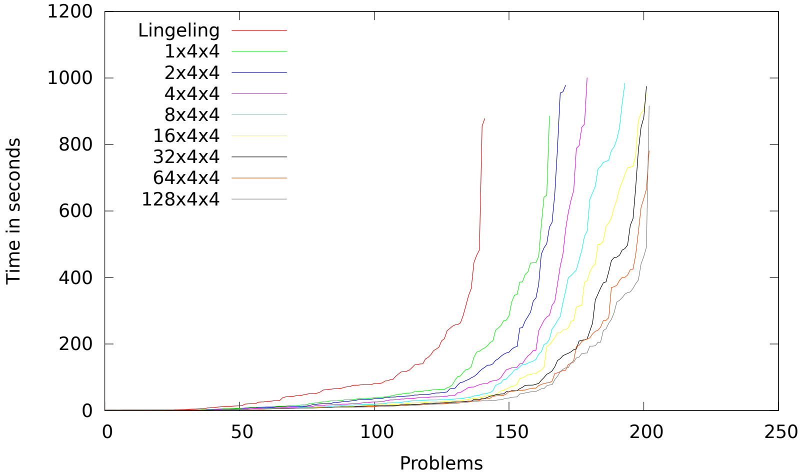
Experiments – (P)lingeling Comparison



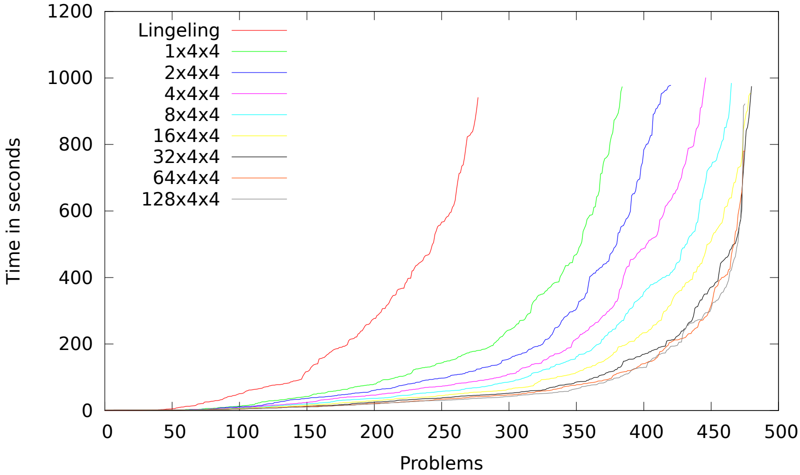
Experiments – (P)lingeling Comparison



Experiments – Scalability on SAT 2011



Experiments – SAT 2011+2014



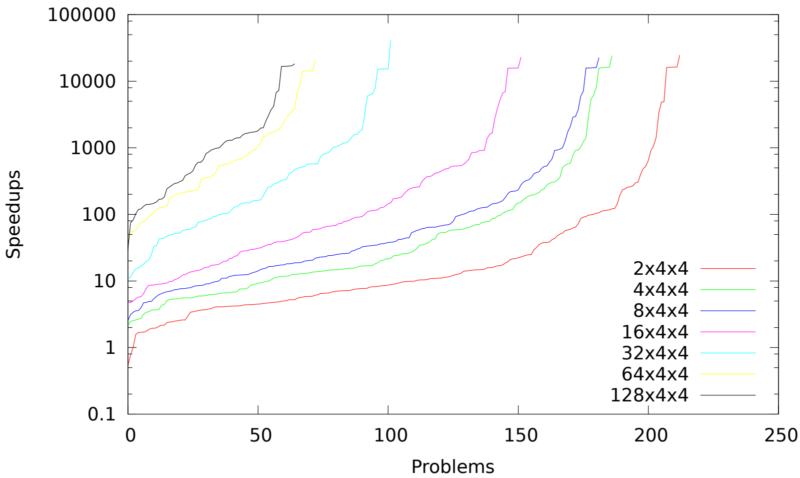
Experiments – Speedups

Big Instance = solved after $10 \cdot (\# threads)$ seconds by Lingeling

Core Solvers	Parallel Solved	Both Solved	Speedup All Avg.	Speedup All Tot.	Speedup All Med.	Speedup Big Avg.	Speedup Big Tot.	Speedup Big Med.
1x4x4	385	363	303	25.01	3.08	524	26.83	4.92
2x4x4	421	392	310	30.38	4.35	609	33.71	9.55
4x4x4	447	405	323	41.30	5.78	766	49.68	16.92
8x4x4	466	420	317	50.48	7.81	801	60.38	32.55
16x4x4	480	425	330	65.27	9.42	1006	85.23	63.75
32x4x4	481	427	399	83.68	11.45	1763	167.13	162.22
64x4x4	476	421	377	104.01	13.78	2138	295.76	540.89
128x4x4	476	421	407	109.34	13.05	2607	352.16	867.00

Experiments – Speedups on Big Inst.

Big Instance = solved after 10 · (#*threads*) seconds by Lingeling



Conclusion

- HordeSat is scalable in highly parallel environments.
- Superlinear and nearly linear scaling in average, total, and median speedups, particularly on hard instances.
- Runtimes of difficult SAT instances are reduced from hours to minutes on commodity clusters
  - This may open up new interactive applications
- On a single machine we match the state-of-the-art performance of Plingeling